CS 880: Advanced Complexity Theory	3/6/06

Lecture 16: Expander Codes

Instructor: Dieter van Melkebeek   Scribe: Matthew Anderson, Michael Kowalczyk

1 Introduction

Recall, last time we discussed several bounds for error correcting codes. We looked several possibility arguments for upper and lower bounds on $R(\delta)$ for binary codes. All of the explicit codes we have seen so far have been located on the boundary points (on axis). The construction of binary codes for internal points were not explicit and may be inefficient.

Today we will we look at one additional classical binary code that is has parameters that are not on the boundary. Secondly, we will begin discussing expander codes with linear time decoding and quadratic time encoding (and improve this result next class).

2 Justesen Code

The Justesen Code begins as a Reed-Solomon code. We will work over a field F, with $n = |F|$ elements. We will use polynomials of degree linear in the size of F. Then the encoding of a word is $f(z_1), f(z_2), ..., f(z_n)$ where z_i is the i^{th} element of F. Then $r, \delta > 0$ when viewed as a code over F, this is because a degree d polynomial can have at most d zeroes.

Let's look at several ideas to turn this into a binary code:

- Idea 0: Naive conversion to binary
  Rewrite each z_i in its binary representation taking $\log n$ bits each. This will not work well because now we are worried about single bit errors and the relative distance of code words from zero is not guaranteed to be good.

- Idea 1: Concatenate a linear code
  Concatenate the binary representation with a good linear code on messages of length $k' = \log n$ to the bit sequence. This will require super-log random bits and thus take too much time to compute. To improve this we should use fewer random bits (derandomize).

- Idea 2: Derandomize using a small sample space
  Consider a class of codes C_z:

  $$C_z : F \to F^2 : x \to (x, z \cdot x) \tag{1}$$

  The number of choices for z is n and only $\log n$ bits specifies each code. Let's compute the probability that C_z is bad:

$$\Pr_z[C_z \text{ has a non-zero code word of relative weight} \leq \delta'] \leq 2^{H(\delta')\cdot 2k'} \cdot 2^{-k'} = 2^{(2H(\delta')-1)k'} \leq \epsilon \tag{2}$$

The first term is the number of words in a ball of radius $\delta' \cdot 2k'$. The second term comes from noticing that the code word $(x, y) \in C_z$ occurs for exactly one z because $y/x = z$ and x has a unique inverse (ignoring the cases were either x or z is zero).

In order to decode we need to search over z to find the appropriate C_z. For every n we need to find another z. Also the number of codes in each class is equal to the length of the original code before the conversion to binary.

- Idea 3: Use C_{z_i} to encode $f(z_i)$

  This seems reasonable, but when can it go wrong?

  - $f(z_i) = 0$: we have $\delta > 0$ so even if this happens it will be infrequent.
  - C_{z_i} is not good: this is also unlikely we have just shown a bound on this in the previous section.

  This will give us a code with relative distance $\geq (\delta - \epsilon) \cdot \delta'$ and a rate of $\frac{r}{2}$ (the size of each block is doubled). This coding scheme is efficient and very explicit. Though we do not have time to discuss this it can be made faster using irreducible polynomials over F.

This concludes our discussion of non-expander codes.

3 Expander Codes

There are several different ways to use expanders to construct good codes:

- Construct parity check matrix - this gives linear decoding and quadratic encoding.
- Construct generator matrix - this gives linear decoding and linear encoding.

In these constructions will will use "unique neighbor" expanders (also known as "lossless" expanders). These expanders have very high degree and there exist explicit constructions for them. We will not cover the constructions for these types of expanders instead we will state their properties.

3.1 The Idea

The expanders we construct are bipartite with $L = N$, $R = M$ with left degree d_L, right degree d_R and $n = |N|$ and $m = |M|$ (Figure 1). The vertices in N represent the bits of our code word. The vertices in M represent the parity check of the bits from N connected to each vertex. If the graph is highly expanding the contributions from each bits should be spread out over a lot of the parity checks. There should not exist a small subset of N that satisfies all parity checks as this would mean that the smallest non-zero code word is not very large.

Choose a graph that is left $(\alpha N, d_L(1-\epsilon))$-expanding, then the rate is at least $(1 - \frac{M}{N})$ and the are at least $\frac{2^n}{2^m}$ code words.

Claim 1. *The relative distance of code words is $\geq \alpha$, provided $\epsilon < \frac{1}{2}$.*

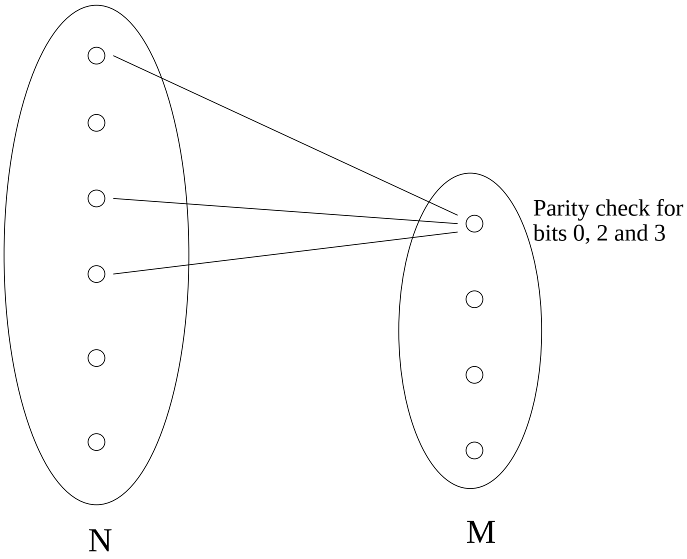

Figure 1: Unique neighbor expander for parity matrix construction

Proof. Consider any word and let B denote the positions of the one bits. If ($|B| \leq \alpha N$) then the expansion property holds. So $|\Gamma(B)| \geq d_L(1-\epsilon) \cdot |B|$. At most $d_L \cdot \epsilon |B|$ vertices in $\Gamma(B)$ do not have a unique neighbor in B (the maximum is $d_L|B|$). So the rest, ($d_L(1-2\epsilon)|B|$), that a unique neighbor in B and those parity checks are unsatisfiable. Choosing $\epsilon < \frac{1}{2}$ will prevent small code words (relative distance $< \alpha$) from existing. □

We hope to be able to correct $\frac{\alpha}{2}$ errors in linear time.

Algorithm 1: Algorithm for decoding received words
Input: Received word R
Output: The decoded word
DECODE(R)
(1) Compute the parity bits of R from the expander.
(2) **while** there are bits in R with a majority of parity checks violated
(3) Select one such bit and flip it.
(4) **return** the final bit settings

Claim 2. *This algorithm runs in linear time.*

Proof. The setup is linear provided we can construct the expander in linear time. There is a constant amount of work to process each bit flip. The number of unsatisfied parity checks must decrease monotonically because we only flip bits that violate a strict majority of their parity checks. Only linearly many parity checks could have been violated so the algorithm will halt and run in linear time. □

Claim 3. *If the input differs from a code word w in at most $\frac{\alpha}{2}$ bit positions then the algorithm returns w, provided that $\epsilon < \frac{1}{4}$.*

Proof. Let

$$B = \{\text{Positions in which the current bit setting differs from } w\}. \tag{3}$$

If ($|B| \leq \alpha \cdot N$) then $|\Gamma(B)| \geq d_L(1-\epsilon)|B|$. There are at least $(1-2\epsilon)d_L|B|$ unique neighbor in $\Gamma(B)$ (by our prior argument). The parity check is not met for the unique neighbors. If ($\epsilon < \frac{1}{4}$) then at least one element of must have a majority of parity checks violated. So the algorithm does not get stuck and will continue to execute until $B = \{\}$.

Either $|B|$ increases or decreases by one each step because exactly one bit is flipped. If ($|B| = \alpha N$) then the number of violated checks is $> \frac{1}{2}d_L \cdot \alpha N$, which is more than the original number of violated checks so $|B| \leq \alpha \cdot N$ must alway hold.

□

Thus the algorithm runs in linear time and produces the decoded word.

An alternate algorithm would be to flip all bits which have a majority of checks violated. The proof that this algorithm also works is slightly more involved. You can show that $|B|$ decreases monotonically by a constant factor allowing only a logarithmic number of steps. This will be a homework problem.

3.2 Existence of Lossless Expanders

Definition 1. *For any positive constants ϵ and ρ, there exists constants d_L and d_R such that for all N you can efficiently construct a N by ρN bipartite graph that is $(K, d_L(1-\epsilon))$-expanding where $K = \Omega(\frac{\rho N}{d_L})$.*

This gives our code a rate of $1-\rho$ and relative distance $\alpha = \frac{K}{N}$. Decoding takes linear time and encoding takes quadratic because the generator matrix is dense for this code.

4 Next Time

Next class we will continue this discussion, exploiting some properties of expanders to provide linear encoding. This will involve recursive construction. The code words will be the original bits of N plus the parity bits. This construction will give us a sparse generator matrix that encodes efficiently.